Intel® oneAPI Collective Communications Library Developer Guide and Reference

Contents

Intel® oneAPI Collective Communications Library

1

Intel® oneAPI Collective Communications Library (oneCCL) provides an efficient implementation of communication patterns used in deep learning.

oneCCL features include:

- Built on top of lower-level communication middleware – Intel® MPI Library and libfabrics.
- Optimized to drive scalability of communication patterns by allowing to easily trade off compute for communication performance.
- Works across various interconnects: InfiniBand*, Cornelis Networks*, and Ethernet.
- Provides common API sufficient to support communication workflows within Deep Learning / distributed frameworks (such as PyTorch*, Horovod*).

oneCCL package comprises the oneCCL Software Development Kit (SDK) and the Intel® MPI Library Runtime components.

Get Started

- Release Notes
- Installation Guide
  - System Requirements
  - Installation using Command Line Interface
  - Find More
- Sample Application
  - Build details
  - Run the sample
- Use oneCCL package from CMake
  - oneCCLConfig files generation

Developer Guide

- Programming Model
  - Host Communication
  - Device Communication
  - Limitations
- General Configuration
  - Execution of Communication Operations
  - Transport Selection
- Advanced Configuration
  - Selection of Collective Algorithms
  - Low-precision Datatypes
  - Caching of Communication Operations
  - Prioritization of Communication Operations
  - Fusion of Communication Operations
  - Enabling OFI/verbs/dmabuf Support

Developer Reference

- oneCCL API
  - Initialization
  - oneCCL Concepts
  - Communication Operations
  - Generic workflow
  - Error Handling

Release Notes

Refer to Intel® oneAPI Collective Communications Library Release Notes.

Installation Guide

This page explains how to install and configure the Intel® oneAPI Collective Communications Library (oneCCL). oneCCL supports different installation scenarios using command line interface.

System Requirements

Visit Intel® oneAPI Collective Communications Library System Requirements to learn about hardware and software requirements for oneCCL.

Installation using Command Line Interface

To install oneCCL using command line interface (CLI), follow these steps:

1. Go to the `ccl` folder:

```
cd ccl
```

2. Create a new folder:

```
mkdir build
```

3. Go to the folder created:

```
cd build
```

4. Launch CMake:

```
cmake ..
```

5. Install the product:

```
make -j install
```

In order to have a clear build, create a new `build` directory and invoke `cmake` within the directory.

Custom Installation

You can customize CLI-based installation (for example, specify directory, compiler, and build type):

- To specify **installation directory**, modify the `cmake` command:

```
cmake .. -DCMAKE_INSTALL_PREFIX=</path/to/installation/directory>
```

  If no `-DCMAKE_INSTALL_PREFIX` is specified, oneCCL is installed into the `_install` subdirectory of the current build directory. For example, `ccl/build/_install`.
- To specify **compiler**, modify the `cmake` command:

```
cmake .. -DCMAKE_C_COMPILER=<c_compiler> -DCMAKE_CXX_COMPILER=<cxx_compiler>
```

- To enable `SYCL` devices communication support, specify `SYCL` compiler (only Intel® oneAPI DPC++/C++ Compiler is supported):

```
cmake .. -DCMAKE_C_COMPILER=icx -DCMAKE_CXX_COMPILER=icpx -DCOMPUTE_BACKEND=dpcpp
```

- To specify the **build type**, modify the `cmake` command:

```
cmake .. -DCMAKE_BUILD_TYPE=[Debug|Release]
```

- To enable `make` verbose output to see all parameters used by `make` during compilation and linkage, modify the `make` command as follows:

```
make -j VERBOSE=1 install
```

Find More

- oneCCL Get Started Guide
- oneCCL GitHub Source Code Repository
- oneCCL Documentation

Sample Application

The sample code below shows how to use oneCCL API to perform allreduce communication for SYCL USM memory.

```
#include <iostream>
#include <mpi.h>
#include "oneapi/ccl.hpp"

void mpi_finalize() {
    int is_finalized = 0;
    MPI_Finalized(&is_finalized);

    if (!is_finalized) {
        MPI_Finalize();
    }
}

int main(int argc, char* argv[]) {
    constexpr size_t count = 10 * 1024 * 1024;

    int size = 0;
    int rank = 0;

    ccl::init();

    MPI_Init(nullptr, nullptr);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
    atexit(mpi_finalize);

    auto device_selector = sycl::default_selector_v;
    sycl::queue q(device_selector);
    std::cout << "Running on " << q.get_device().get_info<sycl::info::device::name>() << "\n";

    /* create kvs */
    ccl::shared_ptr_class<ccl::kvs> kvs;
    ccl::kvs::address_type main_addr;
    if (rank == 0) {
        kvs = ccl::create_main_kvs();
        main_addr = kvs->get_address();
        MPI_Bcast((void*)main_addr.data(), main_addr.size(), MPI_BYTE, 0, MPI_COMM_WORLD);
    }
    else {
        MPI_Bcast((void*)main_addr.data(), main_addr.size(), MPI_BYTE, 0, MPI_COMM_WORLD);
        kvs = ccl::create_kvs(main_addr);
    }

    /* create communicator */
    auto dev = ccl::create_device(q.get_device());
    auto ctx = ccl::create_context(q.get_context());
    auto comm = ccl::create_communicator(size, rank, dev, ctx, kvs);

    /* create stream */
    auto stream = ccl::create_stream(q);

    /* create buffers */
    auto send_buf = sycl::malloc_device<int>(count, q);
    auto recv_buf = sycl::malloc_device<int>(count, q);

    /* open buffers and modify them on the device side */
    auto e = q.submit([&](auto& h) {
        h.parallel_for(count, [=](auto id) {
            send_buf[id] = rank + id + 1;
            recv_buf[id] = -1;
        });
    });

    int check_sum = 0;
    for (int i = 1; i <= size; ++i) {
        check_sum += i;
    }

    /* do not wait completion of kernel and provide it as dependency for operation */
    std::vector<ccl::event> deps;
    deps.push_back(ccl::create_event(e));

    /* invoke allreduce */
    auto attr = ccl::create_operation_attr<ccl::allreduce_attr>();
    ccl::allreduce(send_buf, recv_buf, count, ccl::reduction::sum, comm, stream, attr,
deps).wait();

    /* open recv_buf and check its correctness on the device side */
    sycl::buffer<int> check_buf(count);
    q.submit([&](auto& h) {
        sycl::accessor check_buf_acc(check_buf, h, sycl::write_only);
        h.parallel_for(count, [=](auto id) {
```

```
            if (recv_buf[id] != static_cast<int>(check_sum + size * id)) {
                check_buf_acc[id] = -1;
            }
        });
    });

    q.wait_and_throw();

    /* print out the result of the test on the host side */
    {
        sycl::host_accessor check_buf_acc(check_buf, sycl::read_only);
        size_t i;
        for (i = 0; i < count; i++) {
            if (check_buf_acc[i] == -1) {
                std::cout << "FAILED\n";
                break;
            }
        }
        if (i == count) {
            std::cout << "PASSED\n";
        }
    }

    sycl::free(send_buf, q);
    sycl::free(recv_buf, q);
}
```

Build details

1. Build oneCCL with `SYCL` support (only Intel® oneAPI DPC++/C++ Compiler is supported).
2. Set up the library environment.
3. Use the C++ driver with the -fsycl option to build the sample:

```
icpx -o sample sample.cpp -lccl -lmpi -fsycl
```

Run the sample

Intel® MPI Library is required for running the sample. Make sure that MPI environment is set up.

To run the sample, use the following command:

```
mpiexec <parameters> ./sample
```

where `<parameters>` represents optional mpiexec parameters such as node count, processes per node, hosts, and so on.

> **NOTE** Explore the complete list of oneAPI code samples in the oneAPI Samples Catalog. These samples were designed to help you develop, offload, and optimize multiarchitecture applications targeting CPUs, GPUs, and FPGAs.

Use oneCCL package from CMake

`oneCCLConfig.cmake` and `oneCCLConfigVersion.cmake` are included into oneCCL distribution.

With these files, you can integrate oneCCL into a user project with the find_package command. Successful invocation of `find_package(oneCCL <options>)` creates imported target `oneCCL` that can be passed to the target_link_libraries command.

For example:

```
project(Foo)
add_executable(foo foo.cpp)

# Search for oneCCL
find_package(oneCCL REQUIRED)

# Connect oneCCL to foo
target_link_libraries(foo oneCCL)
```

oneCCLConfig files generation

To generate oneCCLConfig files for oneCCL package, use the provided `cmake/scripts/config_generation.cmake` file:

```
cmake [-DOUTPUT_DIR=<output_dir>] -P cmake/script/config_generation.cmake
```

Programming Model

- Host Communication
- Device Communication
- Limitations

> **NOTE** Check out oneCCL specification that oneCCL is based on.

Host Communication

The communication operations between processes are provided by Communicator.

The example below demonstrates the main concepts of communication on host memory buffers.

Example

Consider a simple oneCCL `allreduce` example for CPU.

1. Create a communicator object with user-supplied size, rank, and key-value store:

```
auto ccl_context = ccl::create_context();
auto ccl_device = ccl::create_device();

auto comms = ccl::create_communicators(
   size,
   vector_class<pair_class<size_t, device>>{ { rank, ccl_device } },
   ccl_context,
   kvs);
```

   Or for convenience use non-vector form without device and context parameters.

```
auto comm = ccl::create_communicator(size, rank, kvs);
```

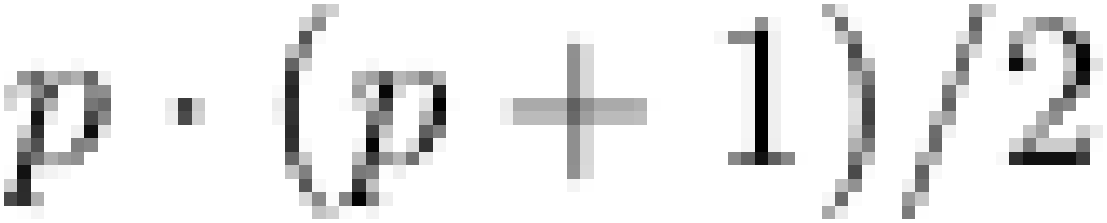
2. Initialize `send_buf` (in real scenario it is supplied by the user):

```
const size_t elem_count = <N>;

/* initialize send_buf */
```

```
for (idx = 0; idx < elem_count; idx++) {
   send_buf[idx] = rank + 1;
}
```

3. `allreduce` invocation performs the reduction of values from all the processes and then distributes the result to all the processes. In this case, the result is an array with `elem_count` elements, where all elements are equal to the sum of arithmetical progression:



```
ccl::allreduce(send_buf,
               recv_buf,
               elem_count,
               reduction::sum,
               comm).wait();
```

4. Check the correctness of `allreduce` operation:

```
auto comm_size = comm.size();
auto expected = comm_size * (comm_size + 1) / 2;

for (idx = 0; idx < elem_count; idx++) {
   if (recv_buf[idx] != expected) {
         std::count << "unexpected value at index " << idx << std::endl;
         break;
   }
}
```

Device Communication

The communication operations between devices are provided by Communicator.

The example below demonstrates the main concepts of communication on device memory buffers.

Example

Consider a simple oneCCL `allreduce` example for GPU:

1. Create oneCCL communicator objects with user-supplied size, rank <-> SYCL device mapping, SYCL context and key-value store:

```
auto ccl_context = ccl::create_context(sycl_context);
auto ccl_device = ccl::create_device(sycl_device);

auto comms = ccl::create_communicators(
   size,
   vector_class<pair_class<size_t, device>>{ { rank, ccl_device } },
   ccl_context,
   kvs);
```

2. Create oneCCL stream object from user-supplied `sycl::queue` object:

```
auto stream = ccl::create_stream(sycl_queue);
```

3. Initialize `send_buf` (in real scenario it is supplied by the user):

```
const size_t elem_count = <N>;

/* using SYCL buffer and accessor */
```

```
auto send_buf_host_acc = send_buf.get_host_access(h, sycl::write_only);
for (idx = 0; idx < elem_count; idx++) {
   send_buf_host_acc[idx] = rank;
}
```

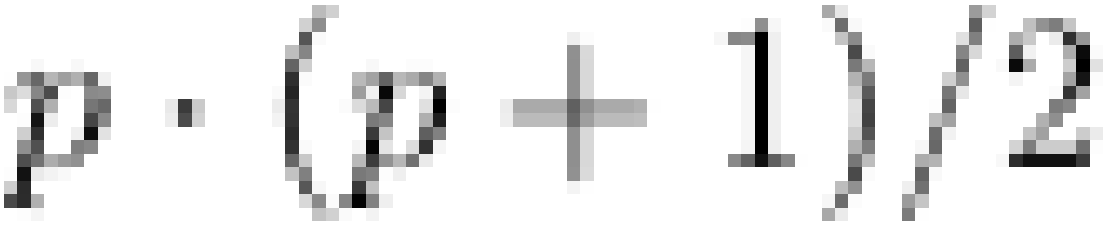
```
/* or using SYCL USM */
for (idx = 0; idx < elem_count; idx++) {
   send_buf[idx] = rank;
}
```

4. For demonstration purposes, modify the `send_buf` on the GPU side:

```
/* using SYCL buffer and accessor */
sycl_queue.submit([&](cl::sycl::handler& h) {
   auto send_buf_dev_acc = send_buf.get_access<mode::write>(h);
   h.parallel_for(range<1>{elem_count}, [=](item<1> idx) {
         send_buf_dev_acc[idx] += 1;
   });
});
```

```
/* or using SYCL USM */
for (idx = 0; idx < elem_count; idx++) {
   send_buf[idx]+ = 1;
}
```

5. `allreduce` invocation performs reduction of values from all processes and then distributes the result to all processes. In this case, the result is an array with `elem_count` elements, where all elements are equal to the sum of arithmetical progression:



```
std::vector<event> events;
for (auto& comm : comms) {
   events.push_back(ccl::allreduce(send_buf,
                                   recv_buf,
                                   elem_count,
                                   reduction::sum,
                                   comm,
                                   streams[comm.rank()]));
}

for (auto& e : events) {
   e.wait();
}
```

6. Check the correctness of `allreduce` operation on the GPU:

```
/* using SYCL buffer and accessor */

auto comm_size = comm.size();
auto expected = comm_size * (comm_size + 1) / 2;

sycl_queue.submit([&](handler& h) {
   auto recv_buf_dev_acc = recv_buf.get_access<mode::write>(h);
   h.parallel_for(range<1>{elem_count}, [=](item<1> idx) {
         if (recv_buf_dev_acc[idx] != expected) {
            recv_buf_dev_acc[idx] = -1;
```

```
        }
    });
});

...

auto recv_buf_host_acc = recv_buf.get_host_access(sycl::read_only);
for (idx = 0; idx < elem_count; idx++) {
    if (recv_buf_host_acc[idx] == -1) {
        std::count << "unexpected value at index " << idx << std::endl;
        break;
    }
}
```

```
/* or using SYCL USM */

auto comm_size = comm.size();
auto expected = comm_size * (comm_size + 1) / 2;

for (idx = 0; idx < elem_count; idx++) {
    if (recv_buf[idx] != expected) {
        std::count << "unexpected value at index " << idx << std::endl;
        break;
    }
}
```

Limitations

The list of scenarios not yet supported by oneCCL:

- Creation of multiple ranks within single process

General Configuration

Execution of Communication Operations

Communication operations are executed by CCL worker threads (workers). The number of workers is controlled by the CCL_WORKER_COUNT environment variable.

Workers affinity is controlled by CCL_WORKER_AFFINITY.

By setting workers affinity you can specify which CPU cores are used by CCL workers. The general rule of thumb is to use different CPU cores for compute (e.g. by specifying `KMP_AFFINITY`) and for CCL communication.

There are two ways to set workers affinity: automatic and explicit.

Automatic setup

To set affinity automatically, set `CCL_WORKER_AFFINITY` to `auto`.

Example

In the example below, oneCCL creates four workers per process and pins them to the last four cores available for the process (available if `mpirun` launcher from oneCCL package is used, the exact IDs of CPU cores depend on the parameters passed to `mpirun`) or to the last four cores on the node.

```
export CCL_WORKER_COUNT=4
export CCL_WORKER_AFFINITY=auto
```

Explicit setup

To set affinity explicitly for all local workers, pass ID of the cores to the `CCL_WORKER_AFFINITY` environment variable.

Example

In the example below, oneCCL creates 4 workers per process and pins them to cores with numbers 3, 4, 5, and 6, respectively:

```
export CCL_WORKER_COUNT=4
export CCL_WORKER_AFFINITY=3,4,5,6
```

Transport Selection

oneCCL supports two transports for inter-process communication: Intel® MPI Library and libfabric*.

The transport selection is controlled by CCL_ATL_TRANSPORT.

In case of MPI over libfabric implementation (for example, Intel® MPI Library 2021) or in case of direct libfabric transport, the selection of specific libfabric provider is controlled by the `FI_PROVIDER` environment variable.

Advanced Configuration

- Selection of Collective Algorithms
- Low-precision Datatypes
- Caching of Communication Operations
- Prioritization of Communication Operations
- Fusion of Communication Operations
- Enabling OFI/verbs/dmabuf Support

Selection of Collective Algorithms

oneCCL supports manual selection of collective algorithms for different message size ranges.

Refer to Collective Algorithms Selection section for details.

Low-precision Datatypes

oneCCL provides support for collective operations on low-precision (LP) datatypes (bfloat16 and float16).

Reduction of LP buffers (for example as phase in `ccl::allreduce`) includes conversion from LP to FP32 format, reduction of FP32 values and conversion from FP32 to LP format.

oneCCL utilizes CPU vector instructions for FP32 <-> LP conversion.

For BF16 <-> FP32 conversion oneCCL provides `AVX512F` and `AVX512_BF16`-based implementations. `AVX512F`-based implementation requires GCC 4.9 or higher. `AVX512_BF16`-based implementation requires GCC 10.0 or higher and GNU binutils 2.33 or higher. `AVX512_BF16`-based implementation may provide less accuracy loss after multiple up-down conversions.

For FP16 <-> FP32 conversion oneCCL provides `F16C` and `AVX512F`-based implementations. Both implementations require GCC 4.9, Clang 9.0 or higher.

utilizes CPU vector instructions for LP numeric operations.

For FP16 numeric operations (arithmetic, load, store) provides `AVX512FP16`-based implementation. This implementation requires GCC 12.0, Clang 14.0, Intel 2021.4.0 or higher.

Refer to Low-precision datatypes for details about relevant environment variables.

Caching of Communication Operations

Communication operations may have expensive initialization phase (for example, allocation of internal structures and buffers, registration of memory buffers, handshake with peers, and so on). oneCCL amortizes these overheads by caching operation internal representations and reusing them on the subsequent calls.

To control this, use operation attribute and set `true` value for `to_cache` field and unique string (for example, tensor name) for `match_id` field.

Note that:

- `match_id` should be the same for a specific communication operation across all ranks.
- If the same tensor is a part of different communication operations, `match_id` should have different values for each of these operations.

Prioritization of Communication Operations

oneCCL supports prioritization of communication operations that controls the order in which individual communication operations are executed. This allows to postpone execution of non-urgent operations to complete urgent operations earlier, which may be beneficial for many use cases.

The communication prioritization is controlled by priority value. Note that the priority must be a non-negative number with a higher number standing for a higher priority.

There are the following prioritization modes:

- None - default mode when all communication operations have the same priority.
- Direct - you explicitly specify priority using `priority` field in operation attribute.
- LIFO (Last In, First Out) - priority is implicitly increased on each operation call. In this case, you do not have to specify priority.

The prioritization mode is controlled by CCL_PRIORITY.

Fusion of Communication Operations

In some cases, it may be beneficial to postpone execution of communication operations and execute them all together as a single operation in a batch mode. This can reduce operation setup overhead and improve interconnect saturation.

oneCCL provides several knobs to enable and control such optimization:

- The fusion is enabled by CCL_FUSION.
- The advanced configuration is controlled by:
  - CCL_FUSION_BYTES_THRESHOLD
  - CCL_FUSION_COUNT_THRESHOLD

- CCL_FUSION_CYCLE_MS

NOTE For now, this functionality is supported for `allreduce` operations only.

Enabling OFI/verbs/dmabuf Support

oneCCL provides experimental support for data transfers between Intel GPU memory and NIC using Linux dmabuf, which is exposed through OFI API for verbs provider.

Requirements

- Linux kernel version >= 5.12
- RDMA core version >= 34.0
- level-zero-devel package

Usage

oneCCL, OFI and OFI/verbs from Intel® oneAPI Base Toolkit support device memory transfers. Refer to Run instructions for usage.

If you want to build software components from sources, refer to Build instructions.

Build instructions

OFI

```
git clone --single-branch --branch v1.13.2 https://github.com/ofiwg/libfabric.git
cd libfabric
./autogen.sh
./configure --prefix=<ofi_install_dir> --enable-verbs=<rdma_core_install_dir> --with-
ze=<level_zero_install_dir> --enable-ze-dlopen=yes
make -j install
```

NOTE You may also get OFI release package directly from here. No need to run autogen.sh if using the release package.

oneCCL

```
cmake -DCMAKE_INSTALL_PREFIX=<ccl_install_dir> -DLIBFABRIC_DIR=<ofi_install_dir> -
DCMAKE_C_COMPILER=icx -DCMAKE_CXX_COMPILER=icpx -DCOMPUTE_BACKEND=dpcpp -DENABLE_OFI_HMEM=1 ..
make -j install
```

Run instructions

1. Set the environment. See Get Started Guide.
2. Run allreduce test with ring algorithm and SYCL USM device buffers:

```
export CCL_ATL_TRANSPORT=ofi
export CCL_ATL_HMEM=1
export CCL_ALLREDUCE=ring
export FI_PROVIDER=verbs
mpiexec -n 2 <ccl_install_dir>/examples/sycl/sycl_allreduce_usm_test gpu device
```

oneCCL API

Generic workflow

Refer to oneCCL specification for more details about generic workflow with oneCCL API.

Error Handling

Refer to oneCCL specification for more details about error handling.

Initialization

template<class... attr_val_type> init_attr CCL_API create_init_attr (attr_val_type &&... avs)

Creates an attribute object that may be used to control the init operation.

Returns	an attribute object

void CCL_API init (const init_attr &attr=default_init_attr)

Initializes the library. Optional for invocation.

Parameters	**attr** – optional init attributes

library_version CCL_API get_library_version ()

Retrieves the library version.

oneCCL Concepts

Refer to oneCCL specification for more details about oneCCL **main concepts**.

- Communicator
- Context
- Device
- Event
- Key-value Store
- Stream

Communicator

template<class... attr_val_type> comm_attr CCL_API create_comm_attr (attr_val_type &&... avs)

Creates an attribute object that may be used to control the create_communicator operation.

Returns an attribute object

template<class... attr_val_type> comm_split_attr CCL_API create_comm_split_attr (attr_val_type &&... avs)

Creates an attribute object that may be used to control the split_communicator operation.

Returns an attribute object

template<class DeviceType, class ContextType> vector_class< communicator > CCL_API create_communicators (int size, const vector_class< pair_class< int, DeviceType >> &devices, const ContextType &context, shared_ptr_class< kvs_interface > kvs, const comm_attr &attr=default_comm_attr)

Creates new communicators with user supplied size, ranks, local device-rank mapping and kvs.

Parameters
- **size** – user-supplied total number of ranks
- **rank** – user-supplied rank
- **device** – local device
- **devices** – user-supplied mapping of local ranks on devices
- **context** – context containing the devices
- **kvs** – key-value store for ranks wire-up
- **attr** – optional communicator attributes

Returns vector of communicators / communicator

template<class DeviceType, class ContextType> vector_class< communicator > CCL_API create_communicators (int size, const map_class< int, DeviceType > &devices, const ContextType &context, shared_ptr_class< kvs_interface > kvs, const comm_attr &attr=default_comm_attr)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

template<class DeviceType, class ContextType> communicator CCL_API create_communicator (int size, int rank, DeviceType &device, const ContextType &context, shared_ptr_class< kvs_interface > kvs, const comm_attr &attr=default_comm_attr)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

communicator CCL_API create_communicator (int size, int rank, shared_ptr_class< kvs_interface > kvs, const comm_attr &attr=default_comm_attr)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

template<class DeviceType, class ContextType> vector_class< communicator > CCL_API create_communicators (int size, const vector_class< DeviceType > &devices, const ContextType &context, shared_ptr_class< kvs_interface > kvs, const comm_attr &attr=default_comm_attr)

Creates a new communicators with user supplied size, local devices and kvs. Ranks will be assigned automatically.

Parameters
- **size** – user-supplied total number of ranks
- **devices** – user-supplied device objects for local ranks
- **context** – context containing the devices
- **kvs** – key-value store for ranks wire-up

- **attr** – optional communicator attributes

Returns vector of communicators / communicator

communicator CCL_API create_communicator (int size, shared_ptr_class< kvs_interface > kvs, const comm_attr &attr=default_comm_attr)

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

communicator CCL_API create_communicator (const comm_attr &attr=default_comm_attr)

Creates a new communicator with externally provided size, rank and kvs. Implementation is platform specific and non portable.

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Parameters **attr** – optional communicator attributes

Returns communicator

vector_class< communicator > CCL_API split_communicators (const vector_class< pair_class< communicator, comm_split_attr >> &attrs)

Splits communicators according to attributes.

Parameters **attrs** – split attributes for local communicators

Returns vector of communicators

Context

template<class native_context_type, class = typename std::enable_if<is_context_supported<native_context_type>()>::type> context CCL_API create_context (native_context_type &&native_context)

Creates a new context from @native_contex_type.

Parameters **native_context** – the existing handle of context

Returns context object

context CCL_API create_context ()

Device

template<class native_device_type, class = typename std::enable_if<is_device_supported<native_device_type>()>::type> device CCL_API create_device (native_device_type &&native_device)

Creates a new device from @native_device_type.

Parameters **native_device** – the existing handle of device

Returns device object

device CCL_API create_device ()

Event

template<class event_type, class = typename std::enable_if<is_event_supported<event_type>()>::type> event CCL_API create_event (event_type &native_event)

Creates a new event from @native_event_type.

Parameters	**native_event** – the existing event
Returns	event object

Key-value Store

template<class... attr_val_type> kvs_attr CCL_API create_kvs_attr (attr_val_type &&... avs)

shared_ptr_class< kvs > CCL_API create_main_kvs (const kvs_attr &attr=default_kvs_attr)

Creates a main key-value store. Its address should be distributed using out of band communication mechanism and be used to create key-value stores on other processes.

Parameters	**attr** – optional kvs attributes
Returns	kvs object

shared_ptr_class< kvs > CCL_API create_kvs (const kvs::address_type &addr, const kvs_attr &attr=default_kvs_attr)

Creates a new key-value store from main kvs address.

Parameters	• **addr** – address of main kvs • **attr** – optional kvs attributes
Returns	kvs object

Stream

template<class native_stream_type, class = typename std::enable_if<is_stream_supported<native_stream_type>()>::type> stream CCL_API create_stream (native_stream_type &native_stream)

Creates a new stream from @native_stream_type.

Parameters	**native_stream** – the existing handle of stream
Returns	stream object

stream CCL_API create_stream ()

Communication Operations

Refer to oneCCL specification for more details about **communication operations**.

- Datatypes
- Collective Operations
  - Allgatherv

Datatypes

template<class... attr_val_type> datatype_attr CCL_API create_datatype_attr (attr_val_type &&... avs)

Creates an attribute object that may be used to register custom datatype.

Returns	an attribute object

datatype CCL_API register_datatype (const datatype_attr &attr)

Registers custom datatype to be used in communication operations.

Parameters	**attr** – datatype attributes
Returns	datatype handle

void CCL_API deregister_datatype (datatype dtype)

Deregisters custom datatype.

Parameters	**dtype** – custom datatype handle

size_t CCL_API get_datatype_size (datatype dtype)

Retrieves a datatype size in bytes.

Parameters	**dtype** – datatype handle
Returns	datatype size

Collective Operations

- Allgatherv
- Allreduce
- Alltoall
- Alltoallv
- Barrier
- Broadcast
- Reduce
- ReduceScatter

Operation Attributes

template<class coll_attribute_type, class... attr_val_type> coll_attribute_type CCL_API create_operation_attr (attr_val_type &&... avs)

Creates an attribute object that may be used to customize communication operation.

Returns an attribute object

Allgatherv

event CCL_API allgatherv (const void *send_buf, size_t send_count, void *recv_buf, const vector_class< size_t > &recv_counts, datatype dtype, const communicator &comm, const stream &stream, const allgatherv_attr &attr=default_allgatherv_attr, const vector_class< event > &deps={})

Allgatherv is a collective communication operation that collects data from all the ranks within a communicator into a single buffer. Different ranks may contribute segments of different sizes. The resulting data in the output buffer is the same for each rank.

Parameters
- **send_buf** – the buffer with `send_count` elements of `dtype` that stores local data to be gathered
- **send_count** – the number of elements of type `dtype` in `send_buf`
- **recv_buf** – [out] the buffer to store gathered result of `dtype`, must be large enough to hold values from all ranks, i.e. size should be equal to `dtype` size in bytes * sum of all values in `recv_counts`
- **recv_counts** – array with the number of elements of type `dtype` to be received from each rank
- **dtype** – the datatype of elements in `send_buf` and `recv_buf`
- **comm** – the communicator for which the operation will be performed
- **stream** – abstraction over a device queue constructed via ccl::create_stream
- **attr** – optional attributes to customize operation
- **deps** – an optional vector of the events that the operation should depend on

Returns ccl::event an object to track the progress of the operation

event CCL_API allgatherv (const void *send_buf, size_t send_count, void *recv_buf, const vector_class< size_t > &recv_counts, datatype dtype, const communicator &comm, const allgatherv_attr &attr=default_allgatherv_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

event CCL_API allgatherv (const void *send_buf, size_t send_count, const vector_class< void * > &recv_bufs, const vector_class< size_t > &recv_counts, datatype dtype, const communicator &comm, const stream &stream, const allgatherv_attr &attr=default_allgatherv_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

This overloaded function takes separate receive buffer per rank.

Parameters **recv_bufs** – [out] array of buffers to store gathered result, one buffer per rank; each buffer must be large enough to keep the corresponding `recv_counts` elements of `dtype` size

event CCL_API allgatherv (const void *send_buf, size_t send_count, const vector_class< void * > &recv_bufs, const vector_class< size_t > &recv_counts, datatype dtype, const communicator &comm, const allgatherv_attr &attr=default_allgatherv_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

This overloaded function takes separate receive buffer per rank.

Parameters

recv_bufs – [out] array of buffers to store gathered result, one buffer per rank; each buffer must be large enough to keep the corresponding `recv_counts` elements of `dtype` size

template<class BufferType, class = typename std::enable_if<is_native_type_supported<BufferType>(), event>::type> event CCL_API allgatherv (const BufferType *send_buf, size_t send_count, BufferType *recv_buf, const vector_class< size_t > &recv_counts, const communicator &comm, const stream &stream, const allgatherv_attr &attr=default_allgatherv_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

Parameters

- **send_buf** – the buffer with `send_count` elements of `BufferType` that stores local data to be gathered
- **recv_buf** – [out] the buffer to store gathered result of `BufferType`, must be large enough to hold values from all ranks, i.e. size should be equal to `BufferType` size in bytes * sum of all values in `recv_counts`

template<class BufferType, class = typename std::enable_if<is_native_type_supported<BufferType>(), event>::type> event CCL_API allgatherv (const BufferType *send_buf, size_t send_count, BufferType *recv_buf, const vector_class< size_t > &recv_counts, const communicator &comm, const allgatherv_attr &attr=default_allgatherv_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

Parameters

- **send_buf** – the buffer with `send_count` elements of `BufferType` that stores local data to be gathered
- **recv_buf** – [out] the buffer to store gathered result of `BufferType`, must be large enough to hold values from all ranks, i.e. size should be equal to `BufferType` size in bytes * sum of all values in `recv_counts`

template<class BufferType, class = typename std::enable_if<is_native_type_supported<BufferType>(), event>::type> event CCL_API allgatherv (const BufferType *send_buf, size_t send_count, vector_class< BufferType * > &recv_bufs, const vector_class< size_t > &recv_counts, const communicator &comm, const stream &stream, const allgatherv_attr &attr=default_allgatherv_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

Parameters

- **send_buf** – the buffer with `send_count` elements of `BufferType` that stores local data to be gathered
- **recv_bufs** – [out] array of buffers to store gathered result, one buffer per rank; each buffer must be large enough to keep the corresponding `recv_counts` elements of `BufferType` size

template<class BufferType, class = typename std::enable_if<is_native_type_supported<BufferType>(), event>::type> event CCL_API allgatherv (const BufferType *send_buf, size_t send_count, vector_class< BufferType * > &recv_bufs, const vector_class< size_t > &recv_counts, const communicator &comm, const allgatherv_attr &attr=default_allgatherv_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

Parameters

- **send_buf** – the buffer with `send_count` elements of `BufferType` that stores local data to be gathered
- **recv_bufs** – [out] array of buffers to store gathered result, one buffer per rank; each buffer must be large enough to keep the corresponding `recv_counts` elements of `BufferType` size

template<class BufferObjectType, class = typename std::enable_if<is_class_supported<BufferObjectType>(), event>::type> event CCL_API allgatherv (const BufferObjectType &send_buf, size_t send_count, BufferObjectType &recv_buf, const vector_class< size_t > &recv_counts, const communicator &comm, const stream &stream, const allgatherv_attr &attr=default_allgatherv_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

Parameters

- **send_buf** – the buffer of `BufferObjectType` with `send_count` elements that stores local data to be gathered
- **recv_buf** – [out] the buffer of `BufferObjectType` to store gathered result, must be large enough to hold values from all ranks, i.e. size should be equal to `BufferType` size in bytes * sum of all values in `recv_counts`

template<class BufferObjectType, class = typename std::enable_if<is_class_supported<BufferObjectType>(), event>::type> event CCL_API allgatherv (const BufferObjectType &send_buf, size_t send_count, BufferObjectType &recv_buf, const vector_class< size_t > &recv_counts, const communicator &comm, const allgatherv_attr &attr=default_allgatherv_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

Parameters

- **send_buf** – the buffer of `BufferObjectType` with `send_count` elements that stores local data to be gathered
- **recv_buf** – [out] the buffer of `BufferObjectType` to store gathered result, must be large enough to hold values from all ranks, i.e. size should be equal to `BufferType` size in bytes * sum of all values in `recv_counts`

template<class BufferObjectType, class = typename std::enable_if<is_class_supported<BufferObjectType>(), event>::type> event CCL_API allgatherv (const BufferObjectType &send_buf, size_t send_count, vector_class< reference_wrapper_class< BufferObjectType >> &recv_bufs, const vector_class< size_t > &recv_counts, const communicator &comm, const stream &stream, const allgatherv_attr &attr=default_allgatherv_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

Parameters

- **send_buf** – the buffer of `BufferObjectType` with `send_count` elements that stores local data to be gathered
- **recv_bufs** – [out] array of buffers to store gathered result, one buffer per rank; each buffer must be large enough to keep the corresponding `recv_counts` elements of `BufferObjectType` size

template<class BufferObjectType, class = typename std::enable_if<is_class_supported<BufferObjectType>(), event>::type> event CCL_API allgatherv (const BufferObjectType &send_buf, size_t send_count, vector_class< reference_wrapper_class< BufferObjectType >> &recv_bufs, const vector_class< size_t > &recv_counts, const communicator &comm, const allgatherv_attr &attr=default_allgatherv_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

Parameters

- **send_buf** – the buffer of `BufferObjectType` with `send_count` elements that stores local data to be gathered
- **recv_bufs** – [out] array of buffers to store gathered result, one buffer per rank; each buffer must be large enough to keep the corresponding `recv_counts` elements of `BufferObjectType` size

Allreduce

event CCL_API allreduce (const void *send_buf, void *recv_buf, size_t count, datatype dtype, reduction rtype, const communicator &comm, const stream &stream, const allreduce_attr &attr=default_allreduce_attr, const vector_class< event > &deps={})

Allreduce is a collective communication operation that performs the global reduction operation on values from all ranks of communicator and distributes the result back to all ranks.

Parameters

- **send_buf** – the buffer with `count` elements of `dtype` that stores local data to be reduced
- **recv_buf** – [out] the buffer to store reduced result, must have the same dimension as `send_buf`
- **count** – the number of elements of type `dtype` in `send_buf` and `recv_buf`
- **dtype** – the datatype of elements in `send_buf` and `recv_buf`\`
- **rtype** – the type of the reduction operation to be applied
- **comm** – the communicator for which the operation will be performed
- **stream** – abstraction over a device queue constructed via ccl::create_stream
- **attr** – optional attributes to customize operation
- **deps** – an optional vector of the events that the operation should depend on

Returns

ccl::event an object to track the progress of the operation

event CCL_API allreduce (const void *send_buf, void *recv_buf, size_t count, datatype dtype, reduction rtype, const communicator &comm, const allreduce_attr &attr=default_allreduce_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

template<class BufferType, class = typename std::enable_if<is_native_type_supported<BufferType>(), event>::type> event CCL_API allreduce (const BufferType *send_buf, BufferType *recv_buf, size_t count, reduction rtype, const communicator &comm, const stream &stream, const allreduce_attr &attr=default_allreduce_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

template<class BufferType, class = typename std::enable_if<is_native_type_supported<BufferType>(), event>::type> event CCL_API allreduce (const BufferType *send_buf, BufferType *recv_buf, size_t count, reduction rtype, const communicator &comm, const allreduce_attr &attr=default_allreduce_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

template<class BufferObjectType, class = typename std::enable_if<is_class_supported<BufferObjectType>(), event>::type> event CCL_API allreduce (const BufferObjectType &send_buf, BufferObjectType &recv_buf, size_t count, reduction rtype, const communicator &comm, const stream &stream, const allreduce_attr &attr=default_allreduce_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

template<class BufferObjectType, class = typename std::enable_if<is_class_supported<BufferObjectType>(), event>::type> event CCL_API allreduce (const BufferObjectType &send_buf, BufferObjectType &recv_buf, size_t count, reduction rtype, const communicator &comm, const allreduce_attr &attr=default_allreduce_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

Alltoall

event CCL_API alltoall (const void *send_buf, void *recv_buf, size_t count, datatype dtype, const communicator &comm, const stream &stream, const alltoall_attr &attr=default_alltoall_attr, const vector_class< event > &deps={})

Alltoall is a collective communication operation in which each rank sends distinct equal-sized blocks of data to each rank. The j-th block of `send_buf` sent from the i-th rank is received by the j-th rank and is placed in the i-th block of `recvbuf`.

Parameters

- **send_buf** – the buffer with `count` elements of `dtype` that stores local data to be sent
- **recv_buf** – [out] the buffer to store received result, must be large enough to hold values from all ranks, i.e. at least `comm_size` * `count`
- **count** – the number of elements of type `dtype` to be send to or to received from each rank
- **dtype** – the datatype of elements in `send_buf` and `recv_buf`
- **comm** – the communicator for which the operation will be performed
- **stream** – abstraction over a device queue constructed via ccl::create_stream
- **attr** – optional attributes to customize operation
- **deps** – an optional vector of the events that the operation should depend on

Returns

ccl::event an object to track the progress of the operation

event CCL_API alltoall (const void *send_buf, void *recv_buf, size_t count, datatype dtype, const communicator &comm, const alltoall_attr &attr=default_alltoall_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

event CCL_API alltoall (const vector_class< void * > &send_buf, const vector_class< void * > &recv_buf, size_t count, datatype dtype, const communicator &comm, const stream &stream, const alltoall_attr &attr=default_alltoall_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Parameters
- **send_bufs** – array of buffers with local data to be sent, one buffer per rank
- **recv_bufs** – [out] array of buffers to store received result, one buffer per rank

event CCL_API alltoall (const vector_class< void * > &send_buf, const vector_class< void * > &recv_buf, size_t count, datatype dtype, const communicator &comm, const alltoall_attr &attr=default_alltoall_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Parameters
- **send_bufs** – array of buffers with local data to be sent, one buffer per rank
- **recv_bufs** – [out] array of buffers to store received result, one buffer per rank

template<class BufferType, class = typename std::enable_if<is_native_type_supported<BufferType>(), event>::type> event CCL_API alltoall (const BufferType *send_buf, BufferType *recv_buf, size_t count, const communicator &comm, const stream &stream, const alltoall_attr &attr=default_alltoall_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

template<class BufferType, class = typename std::enable_if<is_native_type_supported<BufferType>(), event>::type> event CCL_API alltoall (const BufferType *send_buf, BufferType *recv_buf, size_t count, const communicator &comm, const alltoall_attr &attr=default_alltoall_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

template<class BufferType, class = typename std::enable_if<is_native_type_supported<BufferType>(), event>::type> event CCL_API alltoall (const vector_class< BufferType * > &send_buf, const vector_class< BufferType * > &recv_buf, size_t count, const communicator &comm, const stream &stream, const alltoall_attr &attr=default_alltoall_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

Parameters
- **send_bufs** – array of buffers with local data to be sent, one buffer per rank
- **recv_bufs** – [out] array of buffers to store received result, one buffer per rank

template<class BufferType, class = typename std::enable_if<is_native_type_supported<BufferType>(), event>::type> event CCL_API alltoall (const vector_class< BufferType * > &send_buf, const vector_class< BufferType * > &recv_buf, size_t count, const communicator &comm, const alltoall_attr &attr=default_alltoall_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

Parameters
- **send_bufs** – array of buffers with local data to be sent, one buffer per rank
- **recv_bufs** – [out] array of buffers to store received result, one buffer per rank

template<class BufferObjectType, class = typename std::enable_if<is_class_supported<BufferObjectType>(), event>::type> event CCL_API alltoall (const BufferObjectType &send_buf, BufferObjectType &recv_buf, size_t count, const communicator &comm, const stream &stream, const alltoall_attr &attr=default_alltoall_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

template<class BufferObjectType, class = typename std::enable_if<is_class_supported<BufferObjectType>(), event>::type> event CCL_API alltoall (const BufferObjectType &send_buf, BufferObjectType &recv_buf, size_t count, const communicator &comm, const alltoall_attr &attr=default_alltoall_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

Parameters
- **send_bufs** – array of buffers with local data to be sent, one buffer per rank
- **recv_bufs** – [out] array of buffers to store received result, one buffer per rank

template<class BufferObjectType, class = typename std::enable_if<is_class_supported<BufferObjectType>(), event>::type> event CCL_API alltoall (const vector_class< reference_wrapper_class< BufferObjectType >> &send_buf, const vector_class< reference_wrapper_class< BufferObjectType >> &recv_buf, size_t count, const communicator &comm, const stream &stream, const alltoall_attr &attr=default_alltoall_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

Parameters
- **send_bufs** – array of buffers with local data to be sent, one buffer per rank
- **recv_bufs** – [out] array of buffers to store received result, one buffer per rank

template<class BufferObjectType, class = typename std::enable_if<is_class_supported<BufferObjectType>(), event>::type> event CCL_API alltoall (const vector_class< reference_wrapper_class< BufferObjectType >> &send_buf, const vector_class< reference_wrapper_class< BufferObjectType >> &recv_buf, size_t count, const communicator &comm, const alltoall_attr &attr=default_alltoall_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

Parameters
- **send_bufs** – array of buffers with local data to be sent, one buffer per rank

- **recv_bufs** – [out] array of buffers to store received result, one buffer per rank

Alltoallv

event CCL_API alltoallv (const void *send_buf, const vector_class< size_t > &send_counts, void *recv_buf, const vector_class< size_t > &recv_counts, datatype dtype, const communicator &comm, const stream &stream, const alltoallv_attr &attr=default_alltoallv_attr, const vector_class< event > &deps={})

Alltoallv is a collective communication operation in which each rank sends distinct blocks of data to each rank. Block sizes may differ. The j-th block of `send_buf` sent from the i-th rank is received by the j-th rank and is placed in the i-th block of `recvbuf`.

Parameters

- **send_buf** – the buffer with elements of `dtype` that stores local blocks to be sent to each rank
- **send_bufs** – array of buffers to store send blocks, one buffer per rank
- **recv_buf** – [out] the buffer to store received result, must be large enough to hold blocks from all ranks
- **recv_bufs** – [out] array of buffers to store receive blocks, one buffer per rank
- **send_counts** – array with the number of elements of type `dtype` in send blocks for each rank
- **recv_counts** – array with the number of elements of type `dtype` in receive blocks from each rank
- **dtype** – the datatype of elements in `send_buf` and `recv_buf`
- **comm** – the communicator for which the operation will be performed
- **stream** – abstraction over a device queue constructed via ccl::create_stream
- **attr** – optional attributes to customize operation
- **deps** – an optional vector of the events that the operation should depend on

Returns

ccl::event an object to track the progress of the operation

event CCL_API alltoallv (const void *send_buf, const vector_class< size_t > &send_counts, void *recv_buf, const vector_class< size_t > &recv_counts, datatype dtype, const communicator &comm, const alltoallv_attr &attr=default_alltoallv_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

event CCL_API alltoallv (const vector_class< void * > &send_bufs, const vector_class< size_t > &send_counts, const vector_class< void * > &recv_bufs, const vector_class< size_t > &recv_counts, datatype dtype, const communicator &comm, const stream &stream, const alltoallv_attr &attr=default_alltoallv_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

event CCL_API alltoallv (const vector_class< void * > &send_bufs, const vector_class< size_t > &send_counts, const vector_class< void * > &recv_bufs, const vector_class< size_t > &recv_counts, datatype dtype, const communicator &comm, const alltoallv_attr &attr=default_alltoallv_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

template<class BufferType, class = typename std::enable_if<is_native_type_supported<BufferType>(), event>::type> event CCL_API alltoallv (const BufferType *send_buf, const vector_class< size_t > &send_counts, BufferType *recv_buf, const vector_class< size_t > &recv_counts, const communicator &comm, const stream &stream, const alltoallv_attr &attr=default_alltoallv_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

template<class BufferType, class = typename std::enable_if<is_native_type_supported<BufferType>(), event>::type> event CCL_API alltoallv (const BufferType *send_buf, const vector_class< size_t > &send_counts, BufferType *recv_buf, const vector_class< size_t > &recv_counts, const communicator &comm, const alltoallv_attr &attr=default_alltoallv_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

template<class BufferType, class = typename std::enable_if<is_native_type_supported<BufferType>(), event>::type> event CCL_API alltoallv (const vector_class< BufferType * > &send_bufs, const vector_class< size_t > &send_counts, const vector_class< BufferType * > &recv_bufs, const vector_class< size_t > &recv_counts, const communicator &comm, const stream &stream, const alltoallv_attr &attr=default_alltoallv_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

template<class BufferType, class = typename std::enable_if<is_native_type_supported<BufferType>(), event>::type> event CCL_API alltoallv (const vector_class< BufferType * > &send_bufs, const vector_class< size_t > &send_counts, const vector_class< BufferType * > &recv_bufs, const vector_class< size_t > &recv_counts, const communicator &comm, const alltoallv_attr &attr=default_alltoallv_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

template<class BufferObjectType, class = typename std::enable_if<is_class_supported<BufferObjectType>(), event>::type> event CCL_API alltoallv (const BufferObjectType &send_buf, const vector_class< size_t > &send_counts, BufferObjectType &recv_buf, const vector_class< size_t > &recv_counts, const communicator &comm, const stream &stream, const alltoallv_attr &attr=default_alltoallv_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

template<class BufferObjectType, class = typename std::enable_if<is_class_supported<BufferObjectType>(), event>::type> event CCL_API alltoallv (const BufferObjectType &send_buf, const vector_class< size_t > &send_counts, BufferObjectType &recv_buf, const vector_class< size_t > &recv_counts, const communicator &comm, const alltoallv_attr &attr=default_alltoallv_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

template<class BufferObjectType, class = typename std::enable_if<is_class_supported<BufferObjectType>(), event>::type> event CCL_API alltoallv (const vector_class< reference_wrapper_class< BufferObjectType >> &send_bufs, const vector_class< size_t > &send_counts, const vector_class< reference_wrapper_class< BufferObjectType >> &recv_bufs, const vector_class< size_t > &recv_counts, const communicator &comm, const stream &stream, const alltoallv_attr &attr=default_alltoallv_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

template<class BufferObjectType, class = typename std::enable_if<is_class_supported<BufferObjectType>(), event>::type> event CCL_API alltoallv (const vector_class< reference_wrapper_class< BufferObjectType >> &send_bufs, const vector_class< size_t > &send_counts, const vector_class< reference_wrapper_class< BufferObjectType >> &recv_bufs, const vector_class< size_t > &recv_counts, const communicator &comm, const alltoallv_attr &attr=default_alltoallv_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

Barrier

event CCL_API barrier (const communicator &comm, const stream &stream, const barrier_attr &attr=default_barrier_attr, const vector_class< event > &deps={})

Barrier synchronization is performed across all ranks of the communicator and it is completed only after all the ranks in the communicator have called it.

Parameters	• **comm** – the communicator for which the operation will be performed • **stream** – abstraction over a device queue constructed via ccl::create_stream • **attr** – optional attributes to customize operation • **deps** – an optional vector of the events that the operation should depend on
Returns	ccl::event an object to track the progress of the operation

event CCL_API barrier (const communicator &comm, const barrier_attr &attr=default_barrier_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Broadcast

event CCL_API broadcast (void *buf, size_t count, datatype dtype, int root, const communicator &comm, const stream &stream, const broadcast_attr &attr=default_broadcast_attr, const vector_class< event > &deps={})

Broadcast is a collective communication operation that broadcasts data from one rank of communicator (denoted as root) to all other ranks.

Parameters	• **send_buf** – [in] the buffer with `count` elements of `dtype` serves as send buffer for root

- **recv_buf** – [out] the buffer with `count` elements of `dtype` serves as receive buffer for all ranks
- **count** – the number of elements of type `dtype` in `buf`
- **dtype** – the datatype of elements in `buf`
- **root** – the rank that broadcasts `buf`
- **comm** – the communicator for which the operation will be performed
- **stream** – abstraction over a device queue constructed via ccl::create_stream
- **attr** – optional attributes to customize operation
- **deps** – an optional vector of the events that the operation should depend on

Returns: ccl::event an object to track the progress of the operation

event CCL_API broadcast (void *buf, size_t count, datatype dtype, int root, const communicator &comm, const broadcast_attr &attr=default_broadcast_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

template<class BufferType, class = typename std::enable_if<is_native_type_supported<BufferType>(), event>::type> event CCL_API broadcast (BufferType *buf, size_t count, int root, const communicator &comm, const stream &stream, const broadcast_attr &attr=default_broadcast_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

template<class BufferType, class = typename std::enable_if<is_native_type_supported<BufferType>(), event>::type> event CCL_API broadcast (BufferType *buf, size_t count, int root, const communicator &comm, const broadcast_attr &attr=default_broadcast_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

template<class BufferObjectType, class = typename std::enable_if<is_class_supported<BufferObjectType>(), event>::type> event CCL_API broadcast (BufferObjectType &buf, size_t count, int root, const communicator &comm, const stream &stream, const broadcast_attr &attr=default_broadcast_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

template<class BufferObjectType, class = typename std::enable_if<is_class_supported<BufferObjectType>(), event>::type> event CCL_API broadcast (BufferObjectType &buf, size_t count, int root, const communicator &comm, const broadcast_attr &attr=default_broadcast_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

Reduce

event CCL_API reduce (const void *send_buf, void *recv_buf, size_t count, datatype dtype, reduction rtype, int root, const communicator &comm, const stream &stream, const reduce_attr &attr=default_reduce_attr, const vector_class< event > &deps={})

Reduce is a collective communication operation that performs the global reduction operation on values from all ranks of the communicator and returns the result to the root rank.

Parameters

- **send_buf** – the buffer with `count` elements of `dtype` that stores local data to be reduced
- **recv_buf** – [out] the buffer to store reduced result, must have the same dimension as `send_buf`. Used by the `root` rank only, ignored by other ranks.
- **count** – the number of elements of type `dtype` in `send_buf` and `recv_buf`
- **dtype** – the datatype of elements in `send_buf` and `recv_buf`
- **rtype** – the type of the reduction operation to be applied
- **root** – the rank that gets the result of reduction
- **comm** – the communicator for which the operation will be performed
- **stream** – abstraction over a device queue constructed via ccl::create_stream
- **attr** – optional attributes to customize operation
- **deps** – an optional vector of the events that the operation should depend on

Returns

ccl::event an object to track the progress of the operation

event CCL_API reduce (const void *send_buf, void *recv_buf, size_t count, datatype dtype, reduction rtype, int root, const communicator &comm, const reduce_attr &attr=default_reduce_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

template<class BufferType, class = typename std::enable_if<is_native_type_supported<BufferType>(), event>::type> event CCL_API reduce (const BufferType *send_buf, BufferType *recv_buf, size_t count, reduction rtype, int root, const communicator &comm, const stream &stream, const reduce_attr &attr=default_reduce_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

template<class BufferType, class = typename std::enable_if<is_native_type_supported<BufferType>(), event>::type> event CCL_API reduce (const BufferType *send_buf, BufferType *recv_buf, size_t count, reduction rtype, int root, const communicator &comm, const reduce_attr &attr=default_reduce_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

template<class BufferObjectType, class = typename std::enable_if<is_class_supported<BufferObjectType>(), event>::type> event CCL_API reduce (const BufferObjectType &send_buf, BufferObjectType &recv_buf, size_t count, reduction rtype, int root, const communicator &comm, const stream &stream, const reduce_attr &attr=default_reduce_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

template<class BufferObjectType, class = typename std::enable_if<is_class_supported<BufferObjectType>(), event>::type> event CCL_API reduce (const BufferObjectType &send_buf, BufferObjectType &recv_buf, size_t count, reduction rtype, int root, const communicator &comm, const reduce_attr &attr=default_reduce_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

ReduceScatter

event CCL_API reduce_scatter (const void *send_buf, void *recv_buf, size_t recv_count, datatype dtype, reduction rtype, const communicator &comm, const stream &stream, const reduce_scatter_attr &attr=default_reduce_scatter_attr, const vector_class< event > &deps={})

Reduce-scatter is a collective communication operation that performs the global reduction operation on values from all ranks of the communicator and scatters the result in blocks back to all ranks.

Parameters	• **send_buf** – the buffer with `comm_size` * `count` elements of `dtype` that stores local data to be reduced • **recv_buf** – [out] the buffer to store result block containing `recv_count` elements of type `dtype` • **recv_count** – the number of elements of type `dtype` in receive block • **dtype** – the datatype of elements in `send_buf` and `recv_buf` • **rtype** – the type of the reduction operation to be applied • **comm** – the communicator for which the operation will be performed • **stream** – abstraction over a device queue constructed via ccl::create_stream • **attr** – optional attributes to customize operation • **deps** – an optional vector of the events that the operation should depend on
Returns	ccl::event an object to track the progress of the operation

event CCL_API reduce_scatter (const void *send_buf, void *recv_buf, size_t recv_count, datatype dtype, reduction rtype, const communicator &comm, const reduce_scatter_attr &attr=default_reduce_scatter_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

template<class BufferType, class = typename std::enable_if<is_native_type_supported<BufferType>(), event>::type> event CCL_API reduce_scatter (const BufferType *send_buf, BufferType *recv_buf, size_t recv_count, reduction rtype, const communicator &comm, const stream &stream, const reduce_scatter_attr &attr=default_reduce_scatter_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

template<class BufferType, class = typename std::enable_if<is_native_type_supported<BufferType>(), event>::type> event CCL_API reduce_scatter (const BufferType *send_buf, BufferType *recv_buf, size_t recv_count, reduction rtype, const communicator &comm, const reduce_scatter_attr &attr=default_reduce_scatter_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

template<class BufferObjectType, class = typename std::enable_if<is_class_supported<BufferObjectType>(), event>::type> event CCL_API reduce_scatter (const BufferObjectType &send_buf, BufferObjectType &recv_buf, size_t recv_count, reduction rtype, const communicator &comm, const stream &stream, const reduce_scatter_attr &attr=default_reduce_scatter_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

template<class BufferObjectType, class = typename std::enable_if<is_class_supported<BufferObjectType>(), event>::type> event CCL_API reduce_scatter (const BufferObjectType &send_buf, BufferObjectType &recv_buf, size_t recv_count, reduction rtype, const communicator &comm, const reduce_scatter_attr &attr=default_reduce_scatter_attr, const vector_class< event > &deps={})

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Type-safe version.

Point-To-Point Operations

Point-to-point operations enable direct communication between two specific entities, facilitating data exchange, synchronization, and coordination within a parallel computing environment.

The following point-to-point operations are available in oneCCL:

- `send`
- `recv`

send

`send` is a blocking point-to-point communication operation that transfers data from a designated memory buffer (`buf`) to a specific peer rank.

```
event CCL_API send(void *buf,
          size_t count,
          datatype dtype,
          int peer,
          const communicator &comm,
          const stream &stream,
          const pt2pt_attr &attr = default_pt2pt_attr,
          const vector_class<event> &deps = {});
```

Parameters

- `buf` - A buffer with `dtype` count elements that contains the data to be sent.
- `count` - The number of `dtype` elements in a `buf`.
- `dtype`- The datatype of elements in a `buf`.
- `peer` - A destination rank.
- `comm` - A communicator for which the operation is performed.
- `stream` - A stream associated with the operation.
- `attr` - Optional attributes to customize the operation.
- `deps` - An optional vector of the events, on which the operation should depend.

Returns

`ccl::event` - An object to track the progress of the operation.

```
event CCL_API send(void* buf,
          size_t count,
          datatype dtype,
          int peer,
          const communicator &comm,
          const pt2pt_attr &attr = default_pt2pt_attr,
          const vector_class<event> &deps = {});
```

Below you can find an overloaded member function provided for the convenience. It differs from the above function only in what argument(s) it accepts.

```
template <class BufferType,
    class = typename std::enable_if<is_native_type_supported<BufferType>(), event>::type>
event CCL_API send(BufferType *buf,
          size_t count,
          int peer,
          const communicator &comm,
          const stream &stream,
          const pt2pt_attr &attr = default_pt2pt_attr,
          const vector_class<event>& deps = {});
```

Below you can find an overloaded member function provided for the convenience. It differs from the above function only in what argument(s) it accepts.:

```
event CCL_API send(BufferType *buf,
          size_t count,
          int peer,
          const communicator &comm,
          const pt2pt_attr &attr = default_pt2pt_attr,
          const vector_class<event> &deps = {});
```

Below you can find an overloaded member function provided for the convenience. It differs from the above function only in what argument(s) it accepts.

```
event CCL_API send(BufferObjectType &buf,
          size_t count,
          int peer,
          const communicator &comm,
          const stream &stream,
          const pt2pt_attr &attr = default_pt2pt_attr,
          const vector_class<event> &deps = {});
```

Below you can find an overloaded member function provided for the convenience. It differs from the above function only in what argument(s) it accepts.

```
event CCL_API send(BufferObjectType &buf,
          size_t count,
          int peer,
          const communicator &comm,
          const pt2pt_attr &attr = default_pt2pt_attr,
          const vector_class<event> &deps = {});
```

recv

`recv` is a blocking point-to-point communication operation that receives data from a peer rank in a memory buffer.

```
event CCL_API recv(void *buf,
          size_t count,
          datatype dtype,
```

```
          int peer,
          const communicator &comm,
          const stream &stream,
          const pt2pt_attr &attr = default_pt2pt_attr,
          const vector_class<event> &deps = {});
```

Parameters

- `buf` - A buffer with `dtype` count elements that contains where the data is received.
- `count` - The number of `dtype` elements in a `buf`.
- `dtype`- The datatype of elements in a `buf`.
- `peer` - A source rank.
- `comm` - A communicator for which the operation is performed.
- `dtream` - A stream associated with the operation.
- `attr` - Optional attributes to customize the operation.
- `deps` - An optional vector of the events, on which the operation should depend.

Returns:

`ccl::event` - An object to track the progress of the operation.

```
event CCL_API recv(void *buf,
          size_t count,
          datatype dtype,
          int peer,
          const communicator &comm,
          const pt2pt_attr &attr = default_pt2pt_attr,
          const vector_class<event>& deps = {});
```

Below you can find an overloaded member function provided for the convenience. It differs from the above function only in what argument(s) it accepts.

```
template <class BufferType,
    class = typename std::enable_if<is_native_type_supported<BufferType>(), event>::type>
event CCL_API recv(BufferType *buf,
          size_t count,
          int peer,
          const communicator &comm,
          const stream &stream,
          const pt2pt_attr &attr = default_pt2pt_attr,
          const vector_class<event> &deps = {});
```

Below you can find an overloaded member function provided for the convenience. It differs from the above function only in what argument(s) it accepts.

```
event CCL_API recv(BufferType *buf,
          size_t count,
          int peer,
          const communicator &comm,
          const pt2pt_attr &attr = default_pt2pt_attr,
          const vector_class<event> &deps = {});
```

Below you can find an overloaded member function provided for the convenience. It differs from the above function only in what argument(s) it accepts.

```
event CCL_API recv(BufferObjectType &buf,
          size_t count,
          int peer,
          const communicator &comm,
```

```
          const stream &stream,
          const pt2pt_attr &attr = default_pt2pt_attr,
          const vector_class<event> &deps = {});
```

Below you can find an overloaded member function provided for the convenience. It differs from the above function only in what argument(s) it accepts.

```
event CCL_API recv(BufferObjectType &buf,
          size_t count,
          int peer,
          const communicator &comm,
          const pt2pt_attr &attr = default_pt2pt_attr,
          const vector_class<event> &deps = {});
```

Environment Variables

Collective Algorithms Selection

oneCCL supports collective operations for the host (CPU) memory buffers and device (GPU) memory buffers. In addition, oneCCL has two different paths to support collectives with GPU buffers; one directly uses Level Zero, and the other uses SYCL. The SYCL path is a new code being developed and not all collectives are supported.

For the Level Zero implementation, in the case of GPU buffers, oneCCL collectives are optimized to execute a hierarchical algorithm composed of an optimized scale-up phase (communication between ranks/processes in the same node) and a scaleout phase (communication between ranks/processes on different nodes). In the case of CPU buffers, the current collective algorithms do not have support for scale-up and scaleout phases; only a non-hierarchical algorithm can be chosen.

With `CCL_<coll_name> = <algo_name>`, you can select the algorithm for the collective in `<coll_name>`. For GPU buffers, the default algorithm is `topo`, which refers to the scale-up algorithm. If you select an algorithm different from `topo`, oneCCL will implement a non-hierarchical algorithm, where it will copy the GPU buffers to the Host (CPU) and will run the specified algorithm.

For CPU buffers, `topo` is not available; you can only select one of the other algorithms in the table for a given collective.

If the collective uses GPU buffers, you can select whether the implementation of the scale-up algorithm should use copy engines or kernels. There is also the option to select the scaleout algorithm using `CCL_<coll_name>_SCALEOUT=<algo_name>`.

Next, environment variables for collective algorithm selection are explained based on the code path (Level Zero or SYCL), the collective being called, and the type of buffer (GPU or CPU).

Level Zero Path (Default)

ALLGATHERV

CCL_ALLGATHERV

Syntax

For the whole message size:

```
CCL_ALLGATHERV=<algo_name>
```

For a specific message size range:

```
CCL_ALLGATHERV="<algo_name_1>[:<size_range_1>][;<algo_name_2>:<size_range_2>][;...]"
```

Where:

- <algo_name> is selected from the list of the available collective algorithms.
- <size_range> is described by the left and the right size borders in the <left>-<right> format. The size is specified in bytes. To specify the maximum message size, use the reserved word max.

Example

```
CCL_ALLGATHERV="direct:0-8192;ring:8193-max"
```

Arguments

<algo_name>	Description
topo	topology-aware algorithm for scale-up. The default for GPU buffers. Not available for CPU buffers.
direct	Based on MPI_Iallgatherv.
naive	Send to all, receive from all.
flat	alltoall-based algorithm.
multi_bcast	Series of broadcast operations with different root ranks.
ring	ring-based algorithm.

Description

Use this environment variable to specify the algorithm for ALLGATHERV.

If using GPU buffers, select CCL_ALLGATHER=topo (the default) to use a hierarchical algorithm for scale-up data transfer across GPUs in the same node. For GPU buffers, when selecting an algorithm different from topo, oneCCL copies the data to the host and follows the specified CPU algorithm.

CCL_ALLGATHERV_MONOLITHIC_PIPELINE_KERNEL

Syntax

```
CCL_ALLGATHERV_MONOLITHIC_PIPELINE_KERNEL=<value>
```

Arguments

<value>	Description
1	Uses compute kernels to transfer data across GPUs for the allgather phase of ALLGATHERV. The default value.
0	Uses copy engines to transfer data across GPUs for the allgather phase of the ALLGATHERV collective.

Description

Set this environment variable to use GPU buffers to specify the scale-up phase of the algorithm for ALLGATHERV. This environment variable allows the user to choose between using compute kernels or copy engines.

This option is only available if CCL_ALLGATHERV=topo (the default for GPU buffers).

CCL_ALLGATHERV_SCALEOUT

Syntax

For the whole message size:

```
CCL_ALLGATHER_SCALEOUT
```

For a specific message size range:

```
CCL_ALLGATHERV_SCALEOUT="<algo_name_1>[:<size_range_1>][;<algo_name_2>:<size_range_2>][;...]"
```

Where:

- <algo_name> is selected from the list of the available scaleout collective algorithms.
- <size_range> is described by the left and the right size borders in the <left>-<right> format. The size is specified in bytes. To specify the maximum message size, use the reserved word max.

Example

```
CCL_ALLGATHERV_SCALEOUT="direct:0-8192;ring:8193-max"
```

Arguments

<algo_name>	Description
direct	Based on MPI_Iallgatherv.
naive	Send to all, receive from all.
flat	alltoall-based algorithm.
multi_bcast	Series of broadcast operations with different root ranks.
ring	ring-based algorithm. The default value.

Description

Set this environment variable to use GPU buffers to specify the scaleout phase of the algorithm for ALLGATHERV. This option is only available if CCL_ALLGATHERV = topo (the default for GPU buffers).

oneCCL internally fills the algorithm selection table with appropriate defaults. Your input complements the selection table.

To see the actual table values, set CCL_LOG_LEVEL=info.

ALLREDUCE

CCL_ALLREDUCE

Syntax

For the whole message size:

```
CCL_ALLREDUCE=<algo_name>
```

For a specific message size range:

```
CCL_ALLREDUCE="<algo_name_1>[:<size_range_1>][;<algo_name_2>:<size_range_2>][;...]"
```

Where:

- <algo_name> is selected from the list of available collective algorithms.
- <size_range> is described by the left and the right size borders in the <left>-<right> format. The size is specified in bytes. To specify the maximum message size, use the reserved word max.

Example

```
CCL_ALLREDUCE="recursive_doubling:0-8192;rabenseifner:8193-1048576;ring:1048577-max"
```

Arguments

<algo_name>	Description
topo	topology-aware algorithm for scale-up. The default for GPU buffers. Not available for CPU buffers.
direct	Based on MPI_Iallreduce.
rabenseifner	Rabenseifner algorithm.
nreduce	May be beneficial for imbalanced workloads.
ring	reduce_scatter + allgather ring. Use CCL_RS_CHUNK_COUNT and CCL_RS_MIN_CHUNK_SIZE to control pipelining on reduce_scatter phase.
double_tree	double-tree algorithm.
recursive_doubling	Recursive doubling algorithm.
2d	Two-dimensional algorithm (reduce_scatter + allreduce + allgather).

Description

Use this environment variable to specify the algorithm for ALLREDUCE.

If using GPU buffers, select CCL_ALLREDUCE=topo (the default) to use a hierarchical algorithm for scale-up data transfer across GPUs in the same node. For GPU buffers, when selecting an algorithm different from topo, oneCCL copies the data to the host and follows the specified CPU algorithm.

oneCCL internally fills the algorithm selection table with appropriate defaults. Your input complements the selection table.

To see the actual table values, set CCL_LOG_LEVEL=info.

CCL_REDUCE_SCATTER_MONOLITHIC_PIPELINE_KERNEL (GPU buffers only)

Syntax

```
CCL_REDUCE_SCATTER_MONOLITHIC_PIPELINE_KERNEL=<value>
```

Arguments

<value>	Description
1	Uses compute kernels to transfer data across GPUs for the reduce-scatter phase of the ALLREDUCE collectives. The default value.
0	Uses copy engines to transfer data across GPUs for the reduce-scatter phase of the ALLREDUCE.

Description

Set this environment variable to use GPU buffers to specify how to perform the reduce_scatter portion of the scale-up ALLREDUCE collective. This variable allows you to choose between using compute kernels or copy engines.

This option is only available if CCL_ALLREDUCE=topo (the default for GPU buffers).

CCL_ALLGATHERV_MONOLITHIC_PIPELINE_KERNEL (GPU buffers only)

Syntax

```
CCL_ALLGATHERV_MONOLITHIC_PIPELINE_KERNEL=<value>
```

Arguments

<value>	Description
1	Uses compute kernels to transfer data across GPUs for the allgather phase of ALLREDUCE. The default value.
0	Uses copy engines to transfer data across GPUs for the allgather phase of the ALLREDUCE collective.

Description

ALLREDUCE is implemented as a reduce-scatter phase followed by an allgather phase.

Set this environment variable to use GPU buffers to specify how to perform the allgather portion of the scale-up ALLREDUCE collective. This environment variable allows the user to choose between using compute kernels or using copy engines. This option is only available if CCL_ALLGATHERV=topo (the default for GPU buffers).

CCL_ALLREDUCE_SCALEOUT (GPU buffers only)

Syntax

For the whole message size:

```
CCL_ALLREDUCE_SCALEOUT=<algo_name>
```

For a specific message size range:

```
CCL_ALLREDUCE_SCALEOUT="<algo_name_1>[:<size_range_1>][;<algo_name_2>:<size_range_2>][;...]"
```

Where:

- <algo_name> is selected from the list of available collective algorithms.
- <size_range> is described by the left and the right size borders the <left>-<right> format. The size is specified in bytes. To specify the maximum message size, use the reserved word max.

Example

```
CCL_ALLREDUCE_SCALEOUT="recursive_doubling:0-8192;rabenseifner:8193-1048576;ring:1048577-max
```

Arguments

direct	Based on MPI_allreduce
rabenseifner	Rabenseifner algorithm.
nreduce	May be beneficial for imbalanced workloads.
ring	reduce_scatter + allgather ring. Use CCL_RS_CHUNK_COUNT and CCL_RS_MIN_CHUNK_SIZE to control pipelining on reduce_scatter phase. The default value.
double_tree	double-tree algorithm.
ring	Recursive doubling algorithm.

Description

Set this environment variable to use GPU buffers to specify the scaleout algorithm for ALLREDUCE. This option is only available if CCL_ALLREDUCE = topo (the default for GPU buffers).

oneCCL internally fills the algorithm selection table with appropriate defaults. Your input complements the selection table.

To see the actual table values, set `CCL_LOG_LEVEL=info`.

ALLTOALL, ALLTOALLV

CCL_ALLTOALL, CCL_ALLTOALLV

Syntax

For the whole message size:

```
CCL_ALLTOALL=<algo_name>  or CCL_ALLTOALLV=<algo_name>
```

For a specific message size range:

```
CCL_ALLTOALL="<algo_name_1>[:<size_range_1>][;<algo_name_2>:<size_range_2>][;...]"
```

or

```
CCL_ALLTOALLV="<algo_name_1>[:<size_range_1>][;<algo_name_2>:<size_range_2>][;...]"
```

Where:

- `<algo_name>` is selected from the list of available collective algorithms.
- `<size_range>` is described by the left and the right size borders in the `<left>-<right>` format. The size is specified in bytes. To specify the maximum message size, use the reserved word max.

Example

```
CCL_ALLTOALL="naive:0-8192;scatter:8193-max"
```

or

```
CCL_ALLTOALLV="naive:0-8192;scatter:8193-max"
```

Arguments

`topo`	topology-aware algorithm. The default for GPU buffers. Not available for CPU buffers.
`direct`	Based on `MPI_Ialltoall`
`naive`	Send to all, receive from all.
`scatter`	scatter-based algorithm.

CCL_ALLTOALLV_MONOLITHIC_KERNEL

Syntax

```
CCL_ALLTOALLV_MONOLITHIC_KERNEL=<value>
```

Arguments

<value>	Description
`1`	Uses compute kernels to transfer data across GPUs for the allgather phase of the `ALLTOALL` and `ALLTOALLV` collectives. The default value.
`0`	Uses copy engines to transfer data across GPUs for the allgather phase of the `ALLTOALL` and `ALLTOALLV` collectives.

Description

Set this environment variable to use GPU buffers to specify the scale-up algorithm for `ALLTOALL` or `ALLTOALLV` This environment variable allows the user to choose between using compute kernels or using copy engines.

This option is only available if `CCL_ALLTOALL=topo` or `CCL_ALLTOALLV=topo`. The default for GPU buffers.

CCL_ALLTOALL_SCALEOUT, CCL_scaleout_ALLTOALLV_scaleout

Syntax

For the whole message size:

```
CCL_ALLTOALL_SCALEOUT=<algo_name>  or CCL_ALLTOALLV_SCALEOUT=<algo_name>
```

For a specific message size range:

```
CCL_ALLTOALL_SCALEOUT="<algo_name_1>[:<size_range_1>][;<algo_name_2>:<size_range_2>][;...]"
```

or

```
CCL_ALLTOALLV_SCALEOUT="<algo_name_1>[:<size_range_1>][;<algo_name_2>:<size_range_2>][;...]"
```

Where:

- `<algo_name>` is selected from the list of available collective algorithms.
- `<size_range>` is described by the left and the right size borders in a format `<left>-<right>`. The size is specified in bytes. To specify the maximum message size, use the reserved word max.

Example

```
CCL_ALLTOALL_SCALEOUT="naive:0-8192;scatter:8193-max"
```

or

```
CCL_ALLTOALLV_SCALEOUT="naive:0-8192;scatter:8193-max"
```

Arguments

<algo_name>	Description
`naive`	Send to all, receive from all.
`scatter`	scatter-based algorithm. The default value.

Description

Set this environment variable to use GPU buffers to specify the scaleout algorithm for `ALLTOALL` or `ALLTOALLV`. This option is only available if `CCL_ALLTOALL=topo` or `CCL_ALLTOALLV=topo` (the default for GPU buffers).

oneCCL internally fills the algorithm selection table with appropriate defaults. Your input complements the selection table.

To see the actual table values, set `CCL_LOG_LEVEL=info`.

BARRIER

CCL_BARRIER

Syntax

```
CCL_BARRIER=<algo_name>
```

Arguments

<algo_name>	Description
direct	Based on MPI_Ibarrier.
ring	Ring-based algorithm.

Description

Use this environment variable to select the barrier algorithm.

BROADCAST

CCL_BCAST

Syntax

```
CCL_BCAST=<algo_name>
```

Arguments

<algo_name>	Description
topo	topology-aware algorithm. The default for GPU buffers. Not available for CPU buffers.
direct	Based on MPI_Ibcast.
ring	ring-based algorithm.
double_tree	double-tree algorithm.
naive	Send to all from root rank.

Description

Use this environment variable to select the algorithm used for broadcast.

> **NOTE** The BCAST algorithm does not yet support the CCL_BCAST_scaleout environment variable. To change the algorithm for BCAST, use the CCL_BCAST environment variable.

REDUCE

CCL_REDUCE

Syntax

For the whole message size:

```
CCL_REDUCE=<algo_name>
```

For a specific message size range:

```
CCL_REDUCE="<algo_name_1>[:<size_range_1>][;<algo_name_2>:<size_range_2>][;...]"
```

Where:

- <algo_name> is selected from the list of available collective algorithms.
- <size_range> is described by the left and the right size borders in the <left>-<right> format. The size is specified in bytes. To specify the maximum message size, use the reserved word max.

Example

```
CCL_REDUCE="direct:0-8192;double_tree:1048577-max"
```

Arguments

<algo_name>	Description
topo	topology-aware algorithm for scale-up. The default for GPU buffers. Not available for CPU buffers.
direct	Based on MPI_Ireduce.
rabenseifner	Rabenseifner algorithm.
tree	tree algorithm
double_tree	double-tree algorithm.

Description

Set this environment variable to specify the algorithm for REDUCE.

If using GPU buffers, select CCL_REDUCE=topo (the default) to use a hierarchical algorithm for scale-up data transfer across GPUs in the same node. For GPU buffers, when selecting an algorithm different from topo, oneCCL copies the data to the host and follows the specified CPU algorithm.

oneCCL internally fills the algorithm selection table with appropriate defaults. Your input complements the selection table.

To see the actual table values, set CCL_LOG_LEVEL=info.

CCL_REDUCE_SCATTER_MONOLITHIC_PIPELINE_KERNEL (GPU buffers only)

Syntax

```
CCL_REDUCE_SCATTER_MONOLITHIC_PIPELINE_KERNEL=<value>
```

Arguments

<value>	Description
1	Uses compute kernels to transfer data across GPUs for the reduce-scatter phase of the REDUCE collective. The default value.
0	Uses copy engines to transfer data across GPUs for the reduce-scatter phase of the REDUCE collective.

Description

Set this environment variable to use GPU buffers to specify the scale-up algorithm for ALLREDUCE. This environment variable allows the user to choose between using compute kernels or using copy engines.

This option is only available if CCL_REDUCE=topo (the default for GPU buffers).

CCL_REDUCE_SCALEOUT (GPU buffers only)

Syntax

For the whole message size:

```
CCL_REDUCE_SCALEOUT=<algo_name>
```

For a specific message size range:

```
CCL_REDUCE_SCALEOUT="<algo_name_1>[:<size_range_1>][;<algo_name_2>:<size_range_2>][;...]"
```

Where:

- <algo_name> is selected from the list of available collective algorithms.
- <size_range> is described by the left and the right size borders in a format <left>-<right>. The size is specified in bytes. To specify the maximum message size, use the reserved word max.

Example

```
CCL_REDUCE_SCALEOUT="direct:0-8192;double_tree:1048577-max"
```

Arguments

<algo_name>	Description
direct	Based on MPI_Ireduce.
rabenseifner	Rabenseifner algorithm.
tree	tree algorithm.
double_tree	double-tree algorithm. The default value.

Description

Set this environment variable to use GPU buffers to specify the scaleout algorithm for REDUCE. This option is only available if CCL_REDUCE=topo (the default for GPU buffers).

oneCCL internally fills the algorithm selection table with appropriate defaults. Your input complements the selection table.

To see the actual table values, set CCL_LOG_LEVEL=info.

REDUCE_SCATTER

CCL_REDUCE_SCATTER

Syntax

For the whole message size:

```
CCL_REDUCE_SCATTER=<algo_name>
```

For a specific message size range:

```
CCL_REDUCE_SCATTER="<algo_name_1>[:<size_range_1>][;<algo_name_2>:<size_range_2>][;...]"
```

Where:

- <algo_name> is selected from the list of available collective algorithms.
- <size_range> is described by the left and the right size borders in a format <left>-<right>. The size is specified in bytes. To specify the maximum message size, use the reserved word max.

Example

```
CCL_REDUCE_SCATTER="direct:0-8192;ring:1048577-max"
```

Arguments

<algo_name>	Description
topo	topology-aware algorithm for scale-up. The default for GPU buffers. Not available for CPU buffers.

direct	Based on MPI_Ireduce_scatter_block.
naive	Send to all, receive, and reduce from all.
ring	ring-based algorithm. Use CCL_RS_CHUNK_COUNT and CCL_RS_MIN_CHUNK_SIZE to control pipelining.

Description

Use this environment variable to specify the algorithm for reduce. If using GPU buffers, select CCL_REDUCE_SCATTER=topo (the default) to use a hierarchical algorithm for scale-up data transfer across GPUs in the same node. For GPU buffers,when selecting an algorithm different from topo, oneCCL copies the data to the host and follow the specified CPU algorithm.

oneCCL internally fills the algorithm selection table with appropriate defaults. Your input complements the selection table.

To see the actual table values, set CCL_LOG_LEVEL=info.

CCL_REDUCE_SCATTER_MONOLITHIC_PIPELINE_KERNEL (GPU buffers only)

Syntax

```
CCL_REDUCE_SCATTER_MONOLITHIC_PIPELINE_KERNEL=<value>
```

Arguments

<value>	Description
1	Uses compute kernels to transfer data across GPUs for the reduce-scatter phase of the REDUCE_SCATTER collective. The default value.
0	Uses copy engines to transfer data across GPUs for the reduce-scatter phase of the REDUCE_SCATTER collective.

Description

Set this environment variable to use GPU buffers to specify how to perform the reduce-scatter portion of the scale-up REDUCE_SCATTER collective. This environment variable allows the user to choose between using compute kernels or using copy engines.

This option is only available if CCL_REDUCE_SCATTER=topo (the default for GPU buffers).

CCL_REDUCE_SCATTER_SCALEOUT (GPU buffers only)

Syntax

For the whole message size:

```
CCL_REDUCE_SCATTER_SCALEOUT=<algo_name>
```

For a specific message size range:

```
CCL_REDUCE_SCATTER_SCALEOUT="<algo_name_1>[:<size_range_1>][;<algo_name_2>:<size_range_2>][;...]"
```

Where:

- <algo_name> is selected from the list of available collective algorithms.
- <size_range> is described by the left and the right size borders in a format <left>-<right>. The size is specified in bytes. To specify the maximum message size, use the reserved word max.

Example

```
CCL_REDUCE_SCATTER_SCALEOUT="direct:0-8192;double_tree:1048577-max"
```

Arguments

<algo_name>	Description
`direct`	Based on `MPI_Ireduce_scatter_block`.
`naive`	Send to all, receive, and reduce from all. The default value.
`ring`	Ring-based algorithm. Use `CCL_RS_CHUNK_COUNT` and `CCL_RS_MIN_CHUNK_SIZE` to control pipelining.

Description

Set this environment variable to use GPU buffers to specify the scaleout algorithm for ALLREDUCE. This option is only available if `CCL_REDUCE_SCATTER = topo` (the default for GPU buffers).

oneCCL internally fills the algorithm selection table with appropriate defaults. Your input complements the selection table.

To see the actual table values, set `CCL_LOG_LEVEL=info`.

SYCL PATH

All collectives

CCL_ENABLE_SYCL_KERNELS

Syntax

```
CCL_ENABLE_SYCL_KERNELS=<value>
```

Arguments

<value>	Description
`1`	Enable SYCL kernels.
`0`	Disable SYCL kernels. The default value.

Description

Setting this environment variable to 1 enables SYCL kernel-based implementations for `ALLGATHERV`, `ALLREDUCE`, and `REDUCE_SCATTER`.

This new optimization optimizes all message sizes and supports the following data types:

- int32
- fp32
- fp16
- bf16
- sum operations
- single nodes

oneCCL falls back to other implementations when the support is unavailable with SYCL kernels, so that you can set up this environment variable safely.

> **NOTE** The name of this variable in 2021.12 was `CCL_SKIP_SCHEDULER`. Starting with 2021.13, the variable has been renamed to `CCL_ENABLE_SYCL_KERNELS`.

Workers

The group of environment variables to control worker threads.

CCL_WORKER_COUNT

Syntax

```
CCL_WORKER_COUNT=<value>
```

Arguments

<value>	Description
N	The number of worker threads for oneCCL rank (1 if not specified).

Description

Set this environment variable to specify the number of oneCCL worker threads.

CCL_WORKER_AFFINITY

Syntax

```
CCL_WORKER_AFFINITY=<cpulist>
```

Arguments

<cpulist>	Description
auto	Workers are automatically pinned to last cores of pin domain. Pin domain depends from process launcher. If mpirun from oneCCL package is used then pin domain is MPI process pin domain. Otherwise, pin domain is all cores on the node.
<cpulist>	A comma-separated list of core numbers and/or ranges of core numbers for all local workers, one number per worker. The i-th local worker is pinned to the i-th core in the list. For example <a>,<b>-<c> defines list of cores containing core with number <a> and range of cores with numbers from <b> to <c>. The core number should not exceed the number of cores available on the system. The length of the list should be equal to the number of workers.

Description

Set this environment variable to specify cpu affinity for oneCCL worker threads.

CCL_WORKER_MEM_AFFINITY

Syntax

```
CCL_WORKER_MEM_AFFINITY=<nodelist>
```

Arguments

<nodelist>	Description
auto	Workers are automatically pinned to NUMA nodes that correspond to CPU affinity of workers.

<nodelist>	Description
<nodelist>	A comma-separated list of NUMA node numbers for all local workers, one number per worker. The i-th local worker is pinned to the i-th NUMA node in the list. The number should not exceed the number of NUMA nodes available on the system.

Description

Set this environment variable to specify memory affinity for oneCCL worker threads.

ATL

The group of environment variables to control ATL (abstract transport layer).

CCL_ATL_TRANSPORT

Syntax

```
CCL_ATL_TRANSPORT=<value>
```

Arguments

<value>	Description
mpi	MPI transport (**default**).
ofi	OFI (libfabric*) transport.

Description

Set this environment variable to select the transport for inter-process communications.

CCL_ATL_HMEM

Syntax

```
CCL_ATL_HMEM=<value>
```

Arguments

<value>	Description
1	Enable heterogeneous memory support on the transport layer.
0	Disable heterogeneous memory support on the transport layer (**default**).

Description

Set this environment variable to enable handling of HMEM/GPU buffers by the transport layer. The actual HMEM support depends on the limitations on the transport level and system configuration.

CCL_ATL_SHM

Syntax

```
CCL_ATL_SHM=<value>
```

Arguments

<value>	Description
0	Disables the OFI shared memory provider. The default value.
1	Enables the OFI shared memory provider.

Description

Set this environment variable to enable the OFI shared memory provider to communicate between ranks in the same node of the host (CPU) buffers. This capability requires OFI as the transport (`CCL_ATL_TRANSPORT=ofi`).

The OFI/SHM provider has support to utilize the Intel(R) Data Streaming Accelerator* (DSA). To run it with DSA*, you need: * Linux* OS kernel support for the DSA* shared work queues * Libfabric* 1.17 or later

To enable DSA, set the following environment variables:

```
FI_SHM_DISABLE_CMA=1
FI_SHM_USE_DSA_SAR=1
```

Refer to Libfabric* Programmer's Manual for the additional details about DSA* support in the SHM provider: https://ofiwg.github.io/libfabric/main/man/fi_shm.7.html.

CCL_PROCESS_LAUNCHER

Syntax

```
CCL_PROCESS_LAUNCHER=<value>
```

Arguments

<value>	Description
hydra	Uses the MPI hydra job launcher. The default value.
torch	Uses a torch job launcher.
pmix	Is used with the PALS job launcher that uses the pmix API. The `mpiexec` command should be similar to: `CCL_PROCESS_LAUNCHER=pmix CCL_ATL_TRANSPORT=mpi mpiexec -np 2 -ppn 2 --pmi=pmix ...`
none	No job launcher is used. You should specify the values for `CCL_LOCAL_SIZE and CCL_LOCAL_RANK`.

Description

Set this environment variable to specify the job launcher.

CCL_LOCAL_SIZE

Syntax

```
CCL_LOCAL_SIZE=<value>
```

Arguments

<value>	Description
SIZE	A total number of ranks on the local host.

Description

Set this environment variable to specify a total number of ranks on a local host.

CCL_LOCAL_RANK

Syntax

```
CCL_LOCAL_RANK=<value>
```

Arguments

<value>	Description
RANK	Rank number of the current process on the local host.

Description

Set this environment variable to specify the rank number of the current process in the local host.

Multi-NIC

CCL_MNIC, CCL_MNIC_NAME and CCL_MNIC_COUNT define filters to select multiple NICs. oneCCL workers will be pinned on selected NICs in a round-robin way.

CCL_MNIC

Syntax

```
CCL_MNIC=<value>
```

Arguments

<value>	Description
global	Select all NICs available on the node.
local	Select all NICs local for the NUMA node that corresponds to process pinning.
none	Disable special NIC selection, use a single default NIC (**default**).

Description

Set this environment variable to control multi-NIC selection by NIC locality.

CCL_MNIC_NAME

Syntax

```
CCL_MNIC_NAME=<namelist>
```

Arguments

<namelist>	Description
<namelist>	A comma-separated list of NIC full names or prefixes to filter NICs. Use the ^ symbol to exclude NICs starting with the specified prefixes. For example, if you provide a list mlx5_0,mlx5_1,^mlx5_2, NICs with the names mlx5_0 and mlx5_1 will be selected, while mlx5_2 will be excluded from the selection.

Description

Set this environment variable to control multi-NIC selection by NIC names.

CCL_MNIC_COUNT

Syntax

```
CCL_MNIC_COUNT=<value>
```

Arguments

<value>	Description
N	The maximum number of NICs that should be selected for oneCCL workers. If not specified then equal to the number of oneCCL workers.

Description

Set this environment variable to specify the maximum number of NICs to be selected. The actual number of NICs selected may be smaller due to limitations on transport level or system configuration.

Inter Process Communication (IPC)

CCL_ZE_CACHE_OPEN_IPC_HANDLES_THRESHOLD

Syntax

```
CCL_ZE_CACHE_OPEN_IPC_HANDLES_THRESHOLD=<value>
```

<value>	Description
N	The number IPC handles in the receiver cache. The default value is 1000.

Description

Use this environment variable to change the number of IPC handles opened with `zeMemOpenIpcHandle()` that oneCCL maintains in its receiving cache. IPC handles refer to Level Zero Memory IPCs.

The IPC handles opened with `zeMemOpenIpcHandle()` are stored by oneCCL in the receiving cache. However, when the number of opened IPC handles exceeds the specified threshold, the cache will evict a handle using a LRU (Last Recently Used) policy. Starting with version 2021.10, the default value is 1000.

CCL_ZE_CACHE_GET_IPC_HANDLES_THRESHOLD

Syntax

```
CCL_ZE_CACHE_GET_IPC_HANDLES_THRESHOLD=<value>
```

<value>	Description
N	The number IPC handles in the receiver cache. The default value is 1000.

Description

Use this environment variable to change the number of IPC handles obtained with `zeMemGetIpcHandle()` that oneCCL maintains in its sender cache. IPC handles refer to Level Zero Memory IPCs.

The IPC handles obtained with `zeMemGetIpcHandle()` are stored by oneCCL in the sender cache. However, when the number of get IPC handles exceeds the specified threshold, the cache will evict a handle using a LRU (Last Recently Used) policy. The default value is 1000.

Low-precision datatypes

The group of environment variables to control processing of low-precision datatypes.

CCL_BF16

Syntax

```
CCL_BF16=<value>
```

Arguments

<value>	Description
avx512f	Select implementation based on AVX512F instructions.
avx512bf	Select implementation based on AVX512_BF16 instructions.

Description

Set this environment variable to select implementation for BF16 <-> FP32 conversion on reduction phase of collective operation. The default value depends on instruction set support on specific CPU. AVX512_BF16-based implementation has precedence over AVX512F-based one.

CCL_FP16

Syntax

```
CCL_FP16=<value>
```

Arguments

<value>	Description
f16c	Select implementation based on F16C instructions.
avx512f	Select implementation based on AVX512F instructions.
avx512fp16	Select implementation based on AVX512FP16 instructions.

Description

Set this environment variable to select implementation for on reduction phase of collective operation. AVX512FP16 uses native FP16 numeric operations for reduction. AVX512F and F16C use FP16 <-> FP32 conversion operations to perform the reduction. The default value depends on instruction set support on specific CPU. AVX512FP16-based implementation has precedence over AVX512F and F16C-based one.

CCL_LOG_LEVEL

Syntax

```
CCL_LOG_LEVEL=<value>
```

Arguments

<value>
error
warn (**default**)
info
debug

<value>
trace

Description

Set this environment variable to control logging level.

CCL_ITT_LEVEL

Syntax

```
CCL_ITT_LEVEL=<value>
```

Arguments

<value>	Description
1	Enable support for ITT profiling.
0	Disable support for ITT profiling (**default**).

Description

Set this environment variable to specify Intel® Instrumentation and Tracing Technology (ITT) profiling level. Once the environment variable is enabled (value > 0), it is possible to collect and display profiling data for oneCCL using tools such as Intel® VTune™ Profiler.

Fusion

The group of environment variables to control fusion of collective operations.

CCL_FUSION

Syntax

```
CCL_FUSION=<value>
```

Arguments

<value>	Description
1	Enable fusion of collective operations
0	Disable fusion of collective operations (**default**)

Description

Set this environment variable to control fusion of collective operations. The real fusion depends on additional settings described below.

CCL_FUSION_BYTES_THRESHOLD

Syntax

```
CCL_FUSION_BYTES_THRESHOLD=<value>
```

Arguments

<value>	Description
SIZE	Bytes threshold for a collective operation. If the size of a communication buffer in bytes is less than or equal to SIZE, then oneCCL fuses this operation with the other ones.

Description

Set this environment variable to specify the threshold of the number of bytes for a collective operation to be fused.

CCL_FUSION_COUNT_THRESHOLD

Syntax

```
CCL_FUSION_COUNT_THRESHOLD=<value>
```

Arguments

<value>	Description
COUNT	The threshold for the number of collective operations. oneCCL can fuse together no more than COUNT operations at a time.

Description

Set this environment variable to specify count threshold for a collective operation to be fused.

CCL_FUSION_CYCLE_MS

Syntax

```
CCL_FUSION_CYCLE_MS=<value>
```

Arguments

<value>	Description
MS	The frequency of checking for collectives operations to be fused, in milliseconds: • Small MS value can improve latency. • Large MS value can help to fuse larger number of operations at a time.

Description

Set this environment variable to specify the frequency of checking for collectives operations to be fused.

CCL_PRIORITY

Syntax

```
CCL_PRIORITY=<value>
```

Arguments

<value>	Description
direct	You have to explicitly specify priority using priority.

<value>	Description
`lifo`	Priority is implicitly increased on each collective call. You do not have to specify priority.
`none`	Disable prioritization (**default**).

Description

Set this environment variable to control priority mode of collective operations.

CCL_MAX_SHORT_SIZE

Syntax

```
CCL_MAX_SHORT_SIZE=<value>
```

Arguments

<value>	Description
`SIZE`	Bytes threshold for a collective operation (`0` if not specified). If the size of a communication buffer in bytes is less than or equal to `SIZE`, then oneCCL does not split operation between workers. Applicable for `ALLREDUCE`, `REDUCE` and `BROADCAST`.

Description

Set this environment variable to specify the threshold of the number of bytes for a collective operation to be split.

CCL_SYCL_OUTPUT_EVENT

Syntax

```
CCL_SYCL_OUTPUT_EVENT=<value>
```

Arguments

<value>	Description
`1`	Enable support for SYCL output event (**default**).
`0`	Disable support for SYCL output event.

Description

Set this environment variable to control support for SYCL output event. Once the support is enabled, you can retrieve SYCL output event from oneCCL event using `get_native()` method. oneCCL event must be associated with oneCCL communication operation.

CCL_ZE_LIBRARY_PATH

Syntax

```
CCL_ZE_LIBRARY_PATH=<value>
```

Arguments

<value>	Description
PATH/NAME	Specify the name and full path to the Level-Zero library for dynamic loading by oneCCL.

Description

Set this environment variable to specify the name and full path to Level-Zero library. The path should be absolute and validated. Set this variable if Level-Zero is not located in the default path. By default oneCCL uses libze_loader.so name for dynamic loading.

Point-To-Point Operations

CCL_RECV

Syntax

```
CCL_RECV=<value>
```

Arguments

<value>	Description
direct	Based on the MPI*/OFI* transport layer.
topo	Uses Intel(R) Xe Link technology across GPUs in a multi-GPU node. The default for GPU buffers.
offload	Based on the MPI*/OFI* transport layer and GPU RDMA when supported by the hardware.

CCL_SEND

Syntax

```
CCL_SEND=<value>
```

Arguments

<value>	Description
direct	Based on the MPI*/OFI* transport layer.
topo	Uses Intel(R) Xe Link technology across GPUs in a multi-GPU node. The default for GPU buffers.
offload	Based on the MPI*/OFI* transport layer and GPU RDMA when supported by the hardware.

CCL_ZE_TMP_BUF_SIZE

Syntax

```
CCL_ZE_TMP_BUF_SIZE=<value>
```

Arguments

<value>	Description
N	Size of the temporary buffer (in bytes) oneCCL uses to perform collective operations with topo algorithm and Level Zero path. Default is 536870912, that is, 512 MBs.

Description

Set this environment variable to change the size of the temporary buffer used by the topo algorithm in the Level Zero path. The value is specified in bytes. The default value is 536870912.

You can tune the value of this variable depending on the system memory available, the memory the application requires, and the message size of the collectives used. With larger values, oneCCL consumes more memory but can provide higher performance. Similarly, small values will reduce memory utilization, but can degrade performance.

Notices and Disclaimers